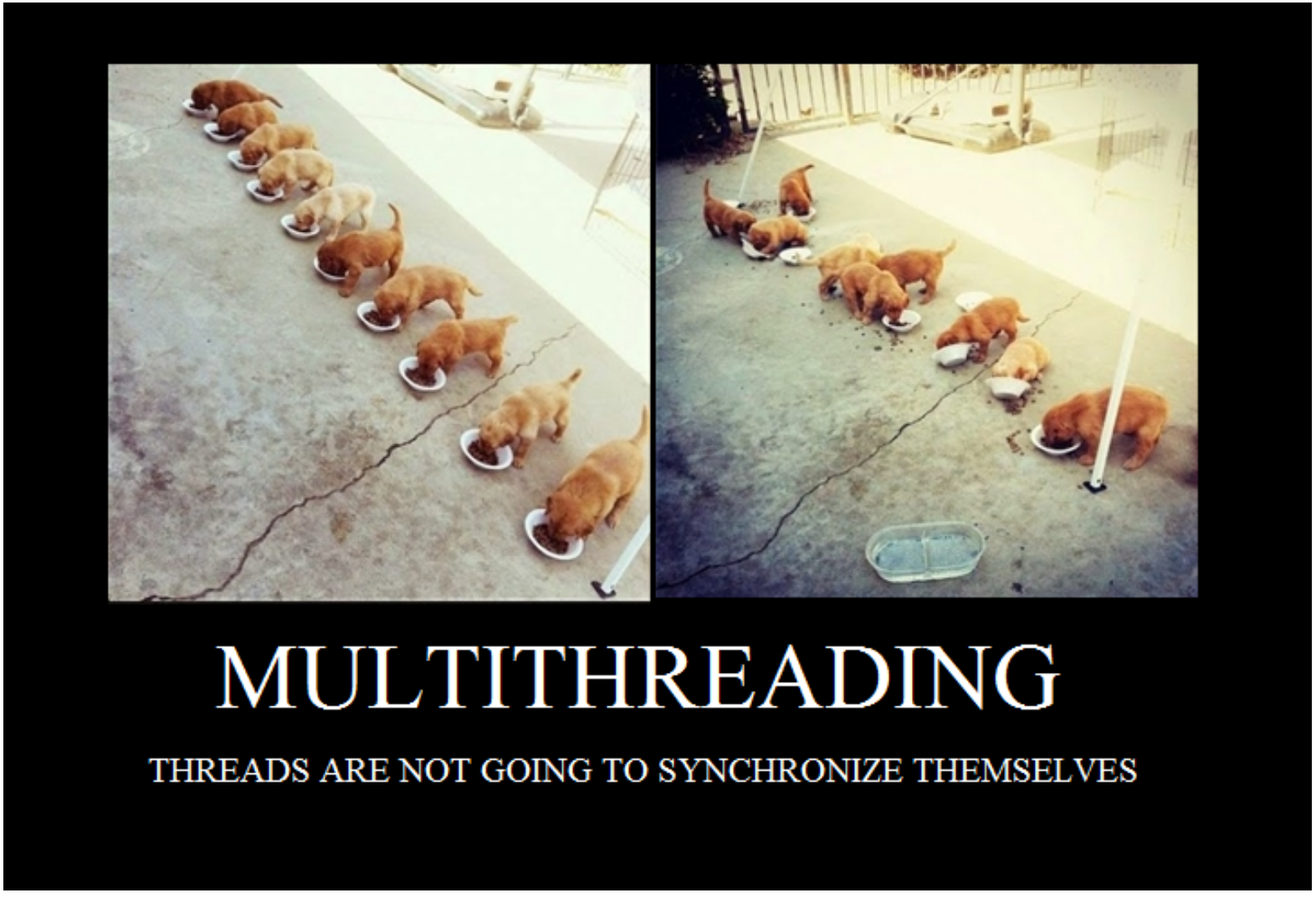
MC504 - Sistemas Operaconais

Sincronização via espera ocupada

Islene Calciolari Garcia
Instituto de Computação - Unicamp
Primeiro Semestre de 2021

Sumário

Multithreading - Expectativa versus Realidade



Tema de hoje: Sincronização via espera ocupada

Mas o que é espera ocupada?

```
ready = 0;
while (!ready) ;
```

- loop potencialmente infinito
- `ready` deve ser alterada por outra thread ou dispositivo
- consome muita CPU enquanto espera :-(

```
$ ./espera_ocupada &
$ top
$ killall ./espera_ocupada
```

- NÃO UTILIZAR ESTA ESTRATÉGIA NO PROJETO 1

Acesso a recursos compartilhados

- Exemplos de recursos a serem compartilhados: estruturas de dados, arquivos, dispositivos etc.
- Estudo de caso simples com variável inteira:

```
volatile int s; /* Variável compartilhada */

/* As threads da aplicação podem ler ou
   escrever na variável s */

s = thr_id;
```

Como assim `volatile`?

- modificador `volatile`
  - indica ao compilador para sempre deixar os valores atualizados em memória
  - não utiliza otimizações via registradores
- modificador `register`
  - sugere ao compilador que uma variável pode ser armazenada em registradores
  - faz sentido hoje em dia???
- veja os códigos `inteiro-simples.c`, `register.c` e `volatile.c`

inteiro-simples.c sem otimização

```
int k;
int main() {
  for (k = 10; k < 100; k++);
  return 0;
}
```

```
$ gcc -O0 -S inteiro-simples.c -o inteiro-simples-O0.s
```

```
        movl $10, k(%rip)
        jmp .L2
.L3:
        movl k(%rip), %eax
        addl $1, %eax
        movl %eax, k(%rip)
.L2:
        movl k(%rip), %eax
        cmpl $99, %eax
        jle .L3
        movl $0, %eax
```

inteiro-simples.c com otimização -O2

```
int k;
int main() {
  for (k = 10; k < 100; k++);
  return 0;
}
```

```
$ gcc -O2 -S inteiro-simples.c -o inteiro-simples-O2.s
```

```
        movl $100, k(%rip)
```

`register.c` sem otimização

```
int main() {
  register int r;
  for (r = 10; r < 100; r++);
  return 0;
}
```

```
$ gcc -O0 -S register.c -o register-O0.s
```

```
        movl $10, %ebx
        jmp .L2
.L3:
        addl $1, %ebx
.L2:
        cmpl $99, %ebx
        jle .L3
```

`register.c` com otimização -O2

```
int main() {
  register int r;
  for (r = 10; r < 100; r++);
  return 0;
}
```

```
$ gcc -O2 -S register.c -o register-O2.s
```

`volatile.c` sem otimização

```
volatile int v;
int main() {
  for (v = 10; v < 100; v++);
  return 0;
}
```

```
$ gcc -O0 -S volatile.c -o volatile-O0.s
```

```
        movl $10, v(%rip)
        jmp .L2
.L3:
        movl v(%rip), %eax
        addl $1, %eax
        movl %eax, v(%rip)
.L2:
        movl v(%rip), %eax
        cmpl $99, %eax
        jle .L3
```

volatile.c com otimização -O2

```
volatile int v;
int main() {
  for (v = 10; v < 100; v++);
  return 0;
}
```

```
$ gcc -O2 -S volatile.c -o volatile-O2.s
```

```
        movl $10, v(%rip)
        movl v(%rip), %eax
        cmpl $99, %eax
        jg .L2
        .p2align 4,,10
        .p2align 3
.L3:
        movl v(%rip), %eax
        addl $1, %eax
        movl %eax, v(%rip)
        movl v(%rip), %eax
        cmpl $99, %eax
        jle .L3
.L2:
```

Acesso à região crítica

- Objetivo: atribuição e impressão sem interferência

```
volatile int s; /* Variável compartilhada */

/* Cada thread tentará executar os seguintes
   comandos sem interferência.   */

s = thr_id;
printf ("Thr %d: %d", thr_id, s);
```

Execução sem interferência

Saída esperada

```
volatile int s; /* Variável compartilhada */
```

Thread 0

(i) `s = 0;`

(ii) `print ("Thr 0:  ", s);`

Thread 1

(iii) `s = 1;`

(iv) `print ("Thr 1:  ", s);`

Saída: Thr 0: 0
Thr 1: 1

Execução sem interferência

Saída esperada II

```
volatile int s; /* Variável compartilhada */
```

	Thread 0		**Thread 1**
(iii)	`s = 0;`	(i)	`s = 1;`
(iv)	`print ("Thr 0:  ", s);`	(ii)	`print ("Thr 1:  ", s);`

Saída: Thr 1: 1
Thr 0: 0

Execução com interferência

Saída inesperada

```
volatile int s; /* Variável compartilhada */
```

	Thread 0		**Thread 1**
(i)	`s = 0;`	(ii)	`s = 1;`
(iii)	`print ("Thr 0:  ", s);`	(iv)	`print ("Thr 1:  ", s);`

Saída: Thr 0: 1
Thr 1: 1

Veja o código: `compartilhada.c`

Exclusão mútua

- Acesso controlado a recursos compartilhados
- Estudo de caso:

```
volatile int s; /* Variável compartilhada */
  while (1) {
      /* Região não crítica */
      /* Protocolo de entrada */
      /* Região crítica */
        s = thr_id;
        printf ("Thr %d: %d", thr_id, s);
      /* Protocolo de saída */
  }
```

Exclusão Mútua

- Os algoritmos devem garantir:
  - exclusão mútua
  - ausência de *deadlock*
  - ausência de *starvation*
  - progresso (uma thread que não esteja interessada na região crítica não pode impedir outra thread de entrar na região crítica)

Observações importantes

- Para fins didáticos, nas análises a seguir vamos supor que as threads executam as operações exatamente na ordem indicada pelo código.
- Na prática, otimizações feitas pelo compilador ou hardware podem alterar esta ordem.
- Para economizar espaço, em alguns slides o modificador `volatile` foi omitido (os códigos estão completos)

Tentando implementar um spin lock

- Lock = variável compartilhada com o seguinte significado:
  - `lock == 0` $\Rightarrow$ região crítica está livre
  - `lock != 0` $\Rightarrow$ região crítica está ocupada
- Protocolo de entrada na região crítica

```
while (lock != 0);
lock = 1;
```

- Protocolo de saída da região crítica

```
lock = 0;
```

Tentando implementar um spin lock

```
volatile int s = 0, lock = 0;
```

Thread 0

```
while (lock != 0);
lock = 1;
s = 0;
print ("Thr 0:" , s);
lock = 0;
```

Thread 1

```
while (lock != 0);
lock = 1;
s = 1;
print ("Thr 1:" , s);
lock = 0;
```

- Veja o código: `tentativa_lock.c`

Solução em hardware

```
entra_RC:
    TSL RX, lock
    CMP RX, #0
    JNE entra_RC
    RET

deixa_RC:
    MOV lock, \#0
    RET
```

- Instrução *test and set* executa atomicamente:
  - lê o conteúdo da variável `lock`;
  - armazena este conteúdo em RX;
  - coloca um valor não nulo em `lock`.
- Não vale para a aula de hoje :-)

Solução em hardware

```
entra_RC:
    MOV RX,#1
    XCHG RX, lock
    CMP RX, #0
    JNE entra_RC
    RET

deixa_RC:
    MOV lock, #0
    RET
```

- Instrução *exchange* troca atomicamente os conteúdos do registrador e da memória;
- Também não vale para a aula de hoje :-)

Abordagem da Alternância

```
int s = 0;
int vez = 1; /* Primeiro a thread 1 */
```

Thread 0

```
while (true)
  while (vez != 0);
  s = 0;
  print ("Thr 0:" , s);
  vez = 1;
```

Thread 1

```
while (true)
  while (vez != 1);
  s = 1;
  print ("Thr 1:" , s);
  vez = 0;
```

- Veja o código: `alternancia.c`

Limitações da Alternância

- Uma thread fora da RC pode impedir outra thread de entrar na RC
- Se uma thread interromper o ciclo a outra não poderá mais entrar na RC

Vetor de Interesse

```
int s = 0;
int interesse[2] = {false, false};
```

Thread 0

```
while (true)
  interesse[0] = true;
  while (interesse[1]);
  s = 0;
  print("Thr 0:" , s);
  interesse[0] = false;
```

Thread 1

```
while (true)
  interesse[1] = true;
  while (interesse[0]);
  s = 1;
  print("Thr 1:" , s);
  interesse[1] = false;
```

- Veja o código: `interesse.c`

Limitações do Vetor de Interesse

- O algoritmo anterior garante exclusão mútua, mas...
- se as duas threads ficarem interessadas ao mesmo tempo haverá *deadlock*.
- Podemos tentar sanar este problema da seguinte forma:

  Se as duas threads ficarem interessadas ao mesmo tempo, elas irão baixar o interesse, esperar um pouco e tentar novamente.
- Veja o próximo código: interesse2.c

Vetor de Interesse e Espera

```
int s = 0;
int interesse[2] = {false, false};
```

Thread 0

```
while (true)
  interesse[0] = true;
  while (interesse[1])
    interesse[0] = false;
    sleep(1);
    interesse[0] = true;
  s = 0;
  print("Thr 0:" , s);
  interesse[0] = false;
```

Thread 1

```
while (true)
  interesse[1] = true;
  while (interesse[0])
    interesse[1] = false;
    sleep(1);
    interesse[1] = true;
  s = 1;
  print("Thr 1:" , s);
  interesse[1] = false;
```

Limitações do Vetor de Interesse e Espera

- O algoritmo anterior garante exclusão mútua, mas...
- se as duas threads andarem sempre no mesmo passo haverá *livelock*.
- Podemos tentar outra abordagem que é:

  Se as duas threads ficarem interessadas ao mesmo tempo, entrará na região crítica a thread cujo identificador estiver marcado na variável vez.

- Veja o próximo código: `interesse_vez.c`

Vetor de Interesse e Alternância

```
int s = 0, vez = 0;
int interesse[2] = {false, false};
```

Thread 0

```
while (true)
  interesse[0] = true;
  if (interesse[1])
    while (vez != 0);
  s = 0;
  print("Thr 0:", s);
  vez = 1;
  interesse[0] = false;
```

Thread 1

```
while (true)
  interesse[1] = true;
  if (interesse[0])
    while (vez != 1);
  s = 1;
  print("Thr 1:", s);
  vez = 0;
  interesse[1] = false;
```

Vetor de Interesse e Alternância

Não garante exclusão mútua

```
int s = 0, vez = 0;
int interesse[2] = {false, false};
```

Thread 0

Thread 1

```
while (true)
  interesse[1] = true;
  if (interesse[0])
    while (vez != 1);
  s = 1;
```

Vetor de Interesse e Alternância

Não garante exclusão mútua

```
int s = 0, vez = 0;
int interesse[2] = {false, false};
```

Thread 0

```
while (true)
  interesse[0] = true;
  if (interesse[1])
    while (vez != 0);
  s = 0;
```

Thread 1

```
while (true)
  interesse[1] = true;
  if (interesse[0])
    while (vez != 1);
  s = 1;
```

Limitações da combinação anterior

- O algoritmo anterior não garante exclusão mútua.
- Podemos tentar melhorar o algoritmo:
  Se as duas threads ficarem interessadas ao mesmo tempo, elas deverão baixar o interesse e só quem tem a vez pode sinalizar o interesse novamente.
- Veja o código: `dekker.c`

Algoritmo de Dekker

```
int s = 0, vez = 0, interesse[2] = {false, false};
```

Thread 0

```
while (true)
  interesse[0] = true;
  while (interesse[1])
    if (vez != 0)
      interesse[0] = false;
      while (vez !=0);
      interesse[0] = true;
  s = 0;
  print ("Thr 0:" , s);
  vez = 1;
  interesse[0] = false;
```

Thread 1

```
while (true)
  interesse[1] = true;
  while(interesse[0])
    if (vez != 1)
      interesse[1] = false;
      while(vez != 1);
      interesse[1] = true;
  s = 1;
  print ("Thr 1:" , s);
  vez = 0;
  interesse[1] = false;
```

Algoritmo de Dekker

```
int s = 0, vez = 0, interesse[2] = {false, false};
```

Thread 0

```
while (true)
  interesse[0] = true;
  while (interesse[1])
```

Thread 1

```
while (true)
  interesse[1] = true;
  while(interesse[0])
    if (vez != 1)
      interesse[1] = false;
      while(vez != 1);
```

Algoritmo de Dekker

```
int s = 0, vez = 0, interesse[2] = {false, false};
```

Thread 0

```
while (true)
  interesse[0] = true;
  while (interesse[1])
    if (vez != 0)
      interesse[0] = false;
      while (vez !=0);
      interesse[0] = true;
  s = 0;
  print ("Thr 0:" , s);
  vez = 1;
  interesse[0] = false;
```

Thread 1

```
while (true)
  interesse[1] = true;
  while(interesse[0])
    if (vez != 1)
      interesse[1] = false;
      while(vez != 1);
```

Próximas aulas

- Algoritmo de Peterson (campeonato)
- Sincronização para N threads
- Semáforos